Université Paris-Dauphine — L3 Informatique

Projet Java — Gestion des Budgets d'une Ville

Titre du rapport :

Simulation municipale et optimisation par sac-à-dos multidimensionnel

(Glouton & Hill Climbing)

Réalisé par :

Zhu Yulei

Ye Kevin

Ghorab Zakaria

Enseignant responsable :

Gilbert Hugo

Année universitaire :

2025 – 2026

Rapport du projet java : Gestion des budgets d'une ville

Introduction

Le projet "Gestion des budgets d'une ville" a pour objectif de simuler la prise de décision d'une équipe municipale dans le choix de projets urbains. Il combine deux axes complémentaires :

- La modélisation organisationnelle d'une équipe municipale (élus, experts, évaluateurs),
- La modélisation algorithmique d'un problème d'optimisation (sac à dos multidimensionnel),
- L'application de deux heuristiques d'optimisation : glouton et hill climbing.

Cette approche permet de relier un problème réel de décision publique à des méthodes informatiques avancées.

Objectif du projet

Simuler le fonctionnement d'une équipe municipale chargée de proposer, évaluer et sélectionner des projets pour
la ville de Dauphine City, selon plusieurs types de coûts (économique, social, environnemental) et de bénéfices.
Le projet inclura aussi une modélisation du problème du sac à dos multidimensionnel et sa résolution heuristique,
les méthodes utilisées seront le glouton et hill climbing.

Arborescence

```
GestionVilleBudget/
│
├── src/
│   ├── equipe/
│   │   ├── Elu.java
│   │   ├── EquipeMunicipale.java
│   │   ├── Evaluateur.java
│   │   ├── EvaluateurRole.java
│   │   ├── Expert.java
│   │   ├── Personne.java
│   │   ├── Projet.java
│   │   ├── Secteur.java
│   │   ├── TestEquipe.java
│   │   └── TypeCout.java
│   │
│   ├── main/
│   │   ├── Main.java
│   │   └── TestAll.java
│   │
│   ├── sacADos/
│   │   ├── Objet.java
```

```
│   │   ├── SacADos.java
│   │   └── VersSacADos.java
│   │
│   ├── solveur/
│   │   ├── glouton/
│   │   │   ├── Comparateurs.java
│   │   │   ├── GloutonAjoutSolver.java
│   │   │   └── GloutonRetraitSolver.java
│   │   │
│   │   └── hillclimbing/
│   │       └── HillClimbingSolver.java
│   │
│
├── bin/
│
├── .settings/          (fichiers VSCode)
├── .classpath
├── .project
```

========== PARTIE SIMULATION EQUIPE MUNICIPALE ==========

Structure du code

Le package “equipe” à été créé pour y incorporer nos différentes classes.
Les classes suivantes ont été intégrée dans le package :

- Elu.java
- EquipeMunicipale.java
- Evaluateur.java
- EvaluateurRole.java
- Expert.java
- Personne.java
- Projet.java
- Secteur.java
- TestEquipe.java
- TypeCout.java

--== Rôles des différentes classes ==--

1. Élu : Évalue le bénéfice d’un projet
2. Equipe Municipale : Regroupe élus, évaluateurs et experts, et gère la simulation
3. Évaluateur : Évalue un coût spécifique (économique, social, environnemental)
4. Expert : Propose des projets selon ses secteurs d’expertise
5. Personnes : Classe de base pour toutes les personnes (classe mère)
6. Projet : Contient les infos d’un projet (titre, description, coûts, bénéfice, secteur)
7. Secteurs : Listes les différentes secteurs

8. Test de l'équipe : Classe de test pour vérifier le fonctionnement de la simulation
9. Type de coûts : Complémentaire pour les types de coûts

========== PARTIE SAC A DOS MULTIDIMENSIONNEL ==========

== OBJECTIF ==

Modéliser le problème du sac à dos multidimensionnel afin de sélectionner des projets en respectant des contraintes budgétaires multiples (économique, social, environnemental). Chaque projet correspond à un "objet" avec : une utilité (bénéfice attendu), plusieurs coûts (3 dimensions), et un ensemble de budgets à ne pas dépasser. Ce modèle sert ensuite de base pour les algorithmes gloutons et hill climbing.

Structure du code

Le package sacADos contient les classes suivantes :
- Objet.java : représente un projet converti en objet pour le sac à dos (utilité + coûts multidimensionnels).
- SacADos.java : représente une instance complète du problème :
  - dimension du problème (nombre de coûts),
  - budgets maximaux,
  - liste des objets disponibles.
- VersSacADos.java : utilitaire permettant la conversion d'une liste de projets en instance de sac à dos.
- SacADosTest.java (si existant) : tests pour vérifier l'admissibilité et les calculs d'utilité.

========== PARTIE RÉSOLVEUR GLOUTONNES ==========

== OBJECTIF ==

Le but du solveur glouton est de fournir rapidement une solution initiale satisfaisante, sans garantie d'optimalité, mais permettant ensuite un point de départ efficace pour le Hill Climbing.
Deux variantes ont été implémentées : glouton par ajout et glouton par retrait.

Structure du code Package :
- solveur.glouton Comparateurs.java Regroupe les différentes heuristiques d'ordre :
  - somme des coûts
  - utilité maximale
  - ratio utilité/coût

- GloutonAjoutSolver.java Ajoute les objets un par un selon un ordre déterminé par un comparateur.
- GloutonRetraitSolver.java Commence par tous les objets, puis retire les moins "intéressants" en premier.

----=== **Principe des méthodes** ===----

--= **Glouton Ajout** =--
1. Trier les objets selon un comparateur.
2. Partir d'une solution vide.
3. Ajouter progressivement chaque objet tant que les budgets ne sont pas dépassés.

--= **Glouton Retrait** =--
1. Démarrer avec tous les objets.
2. Retirer successivement les objets les moins rentables.
3. Une fois admissible, réessayer d'ajouter quelques objets intéressants.

-= **Résultats** =-
Lors des tests effectués dans Main.java, les gloutons renvoient une solution admissible, avec un bénéfice total correct, qui sert ensuite de point de départ pour le Hill Climbing.

========== PARTIE RESOLVEUR HILL CLIMBING ==========

== OBJECTIF ==

Améliorer une solution initiale (souvent fournie par un glouton) en explorant le voisinage des solutions pour augmenter le bénéfice total.
Le Hill Climbing ne garantit pas l'optimum global, mais fournit un optimum local efficace.

Structure du code Package : solveur.hillclimbing

Contient principalement : HillClimbingSolver.java

Cette classe implémente une version déterministe du Hill Climbing :
- point de départ = solution gloutonne
- voisinage = retirer un objet + en ajouter un autre
- accepter uniquement les améliorations
- arrêt lorsqu'aucun voisin ne fournit d'utilité plus élevée

Fonctionnement du Hill Climbing

1. Initialisation Copier la solution initiale.
2. Génération du voisinage Pour chaque objet de la solution : créer un voisin en retirant cet objet, tester l'ajout d'un nouvel objet non présent.
3. Sélection du meilleur voisin admissible calculer l'utilité de chaque voisin, garder le meilleur voisin admissible.
4. Amélioration si le voisin est meilleur : remplacer la solution sinon : arrêter -> optimum local atteint.

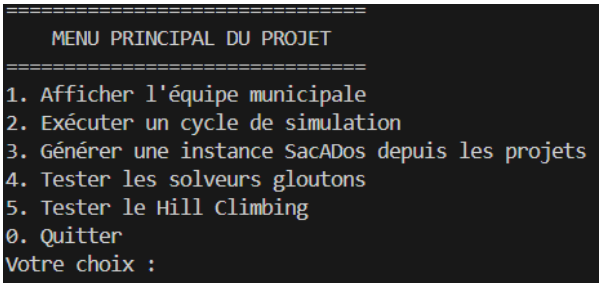
========== PARTIE MAIN ==========

== OBJECTIF ==
La classe Main constitue le point d'entrée du projet.
Elle permet de piloter l'ensemble du système depuis la console à travers un menu interactif :

1. Gestion et affichage de l'équipe municipale,

2. Simulation de propositions de projets,
3. Génération d'une instance de sac à dos multidimensionnel à partir des projets,
4. Test des solveurs gloutons,
5. Test du Hill Climbing

L'objectif est d'offrir une interface simple pour enchaîner toutes les étapes du sujet : de la génération des projets jusqu'à leur sélection via les heuristiques d'optimisation.



== Structure générale ==
La classe Main se trouve dans le package main et contient :

- une méthode main(String[] args) qui lance la boucle de menu
- un Scanner pour lire les entrées utilisateur
- plusieurs méthodes privées qui correspondent aux grandes fonctionnalités du projet :
    1. afficherMenuPrincipal() : affiche les différentes options disponibles
    2. construireEquipeDemonstration() : crée une équipe municipale par défaut (un élu, trois évaluateurs, trois experts avec différents secteurs)
    3. executerSimulation(EquipeMunicipale equipe) : lance un cycle de simulation et affiche les projets générés
    4. creerInstanceDepuisProjets(EquipeMunicipale equipe) : construit une instance de SacADos à partir des projets évalués
    5. testerSolveurs(SacADos sac) : exécute les solveurs gloutons (ajout et retrait)
    6. testerHillClimbing(SacADos sac) : exécute le Hill Climbing à partir d'une solution gloutonne
    7. afficherSolution(String titre, List<Objet> sol, SacADos sac) : affiche proprement une solution (liste d'objets + utilité totale).

== Déroulement du menu ==

Au lancement, le programme :

- Construit une équipe municipale de démonstration via construireEquipeDemonstration() ;
- Initialise une variable instanceSac à null, qui servira plus tard pour le sac à dos.

Ensuite, une boucle while(true) gère le menu principal :

1. *Option 1 : Afficher l'équipe municipale*
   Appelle equipe.afficherEquipe() et permet de vérifier la composition de l'équipe (élu, évaluateurs, experts).

```
==============================
    MENU PRINCIPAL DU PROJET
==============================
1. Afficher l'équipe municipale
2. Exécuter un cycle de simulation
3. Générer une instance SacADos depuis les projets
4. Tester les solveurs gloutons
5. Tester le Hill Climbing
0. Quitter
Votre choix : 1

===== Équipe municipale =====
[ÉLU]      Pierre Martin (45 ans) - Élu(e)

[ÉVALUATEURS]
  - Luc Bernard (38 ans) - Évaluateur (ENVIRONNEMENTAL)
  - Marie Dupont (35 ans) - Évaluateur (ECONOMIQUE)
  - Sophie Durant (40 ans) - Évaluateur (SOCIAL)

[EXPERTS]
  - Jean Leroy (42 ans) - Expert [SPORT, EDUCATION]
  - Claire Moreau (39 ans) - Expert [SANTE, CULTURE]
  - Paul Simon (44 ans) - Expert [ATTRACTIVITE_ECONOMIQUE]
```

2. *Option 2 : Exécuter un cycle de simulation*
   Demande à l'utilisateur le nombre de projets par expert, appelle equipe.executerCycleSimulation(nb) et affiche les projets via equipe.afficherProjets().

```
==============================
    MENU PRINCIPAL DU PROJET
==============================
1. Afficher l'équipe municipale
2. Exécuter un cycle de simulation
3. Générer une instance SacADos depuis les projets
4. Tester les solveurs gloutons
5. Tester le Hill Climbing
0. Quitter
Votre choix : 2
Combien de projets par expert ? 3

===== Projets étudiés =====
1. Projet{titre='Projet EDUCATION #602', secteur=EDUCATION, benefice=58673, coutEco=15564, coutSoc=64983, coutEnv=9287
9}
2. Projet{titre='Projet SPORT #777', secteur=SPORT, benefice=123872, coutEco=46735, coutSoc=68731, coutEnv=54224}
3. Projet{titre='Projet SPORT #442', secteur=SPORT, benefice=132406, coutEco=39886, coutSoc=39183, coutEnv=38772}
4. Projet{titre='Projet SANTE #857', secteur=SANTE, benefice=65522, coutEco=20832, coutSoc=14149, coutEnv=51722}
5. Projet{titre='Projet CULTURE #941', secteur=CULTURE, benefice=114361, coutEco=71634, coutSoc=20859, coutEnv=36452}
6. Projet{titre='Projet SANTE #835', secteur=SANTE, benefice=195731, coutEco=53688, coutSoc=22669, coutEnv=94380}
7. Projet{titre='Projet ATTRACTIVITE_ECONOMIQUE #600', secteur=ATTRACTIVITE_ECONOMIQUE, benefice=139328, coutEco=81443
, coutSoc=64773, coutEnv=45516}
8. Projet{titre='Projet ATTRACTIVITE_ECONOMIQUE #886', secteur=ATTRACTIVITE_ECONOMIQUE, benefice=129949, coutEco=39974
, coutSoc=85564, coutEnv=98716}
9. Projet{titre='Projet ATTRACTIVITE_ECONOMIQUE #891', secteur=ATTRACTIVITE_ECONOMIQUE, benefice=110659, coutEco=62901
, coutSoc=90870, coutEnv=27285}
```

3. *Option 3 : Générer une instance SacADos depuis les projets*
   Vérifie qu'il existe au moins un projet évalué, propose deux modes :
   a. Mode 1 : budgets = coûts (économique, social, environnemental) → utilisation de VersSacADos.depuisProjetSelonCouts(...)
   b. Mode 2 : budgets = secteurs (5 secteurs) → utilisation de VersSacADos.depuisProjetSelonSecteurs(...). stocke le SacADos dans instanceSac.

Mode 1

```
==============================
    MENU PRINCIPAL DU PROJET
==============================
1. Afficher l'équipe municipale
2. Exécuter un cycle de simulation
3. Générer une instance SacADos depuis les projets
4. Tester les solveurs gloutons
5. Tester le Hill Climbing
0. Quitter
Votre choix : 3
Choisir le mode de génération de SacADos :
1. Budgets = coûts (eco, social, env)
2. Budgets = secteurs (5 secteurs)
Votre choix : 1
=== Définir les budgets par type de coût ===
Budget économique : 20000
Budget social : 20000
Budget environnemental : 20000
? Instance SacADos créée (3 dimensions).
```

Mode 2

```
==============================
    MENU PRINCIPAL DU PROJET
==============================
1. Afficher l'équipe municipale
2. Exécuter un cycle de simulation
3. Générer une instance SacADos depuis les projets
4. Tester les solveurs gloutons
5. Tester le Hill Climbing
0. Quitter
Votre choix : 3
Choisir le mode de génération de SacADos :
1. Budgets = coûts (eco, social, env)
2. Budgets = secteurs (5 secteurs)
Votre choix : 2
=== Définir les budgets par secteur ===
Sport : 20000
Santé : 20000
Éducation : 20000
Culture : 20000
Attractivité économique : 20000
? Instance SacADos créée (5 secteurs).
```

4. *Option 4 : Tester les solveurs gloutons*
   Utilise l'instance SacADos créée précédemment puis exécute :
   a. GloutonAjoutSolver avec Comparateurs.f_somme()
   b. GloutonRetraitSolver avec Comparateurs.f_somme() et Comparateurs.f_max()
   affiche les solutions et leur utilité via afficherSolution.

```
==============================
    MENU PRINCIPAL DU PROJET
==============================
1. Afficher l'équipe municipale
2. Exécuter un cycle de simulation
3. Générer une instance SacADos depuis les projets
4. Tester les solveurs gloutons
5. Tester le Hill Climbing
0. Quitter
Votre choix : 4

=== Solveur Glouton AJOUT ===

--- Glouton Ajout ---
  * Objet{ utilite=58673, couts=[0, 0, 15564, 0, 0] }
Utilité totale = 58673

=== Solveur Glouton RETRAIT ===

--- Glouton Retrait ---
  * Objet{ utilite=58673, couts=[0, 0, 15564, 0, 0] }
Utilité totale = 58673
```

5. *Option 5 : Tester le Hill Climbing*
   Commence par une solution gloutonne (résultat de GloutonAjoutSolver), lance le HillClimbingSolver sur cette solution et affiche la solution améliorée et son utilité totale.

```
==============================
    MENU PRINCIPAL DU PROJET
==============================
1. Afficher l'équipe municipale
2. Exécuter un cycle de simulation
3. Générer une instance SacADos depuis les projets
4. Tester les solveurs gloutons
5. Tester le Hill Climbing
0. Quitter
Votre choix : 5

=== Hill Climbing ===

--- Hill Climbing ---
  * Objet{ utilite=58673, couts=[0, 0, 15564, 0, 0] }
Utilité totale = 58673
```

6. *Option 0 : Quitter*
   Sort de la boucle et termine le programme.
   En cas de choix invalide ou si l'utilisateur demande à lancer les solveurs sans avoir d'abord créé une instance de sac à dos, des messages d'erreur explicites sont affichés.

```
==============================
    MENU PRINCIPAL DU PROJET
==============================
1. Afficher l'équipe municipale
2. Exécuter un cycle de simulation
3. Générer une instance SacADos depuis les projets
4. Tester les solveurs gloutons
5. Tester le Hill Climbing
0. Quitter
Votre choix : 0
? Au revoir !
```

========== PARTIE TESTS ==========

Initialisation de l'instance SacADos

```
Instance du sac-à-dos :
  - Objet{ utilite=10, couts=[4, 3, 2] }
  - Objet{ utilite=8, couts=[3, 3, 3] }
  - Objet{ utilite=5, couts=[1, 10, 4] }
  - Objet{ utilite=7, couts=[2, 2, 5] }

Budgets = [7, 7, 7]
```

Test 1 Glouton Ajout (critère : f_somme)
Dans ce premier test, les objets sont triés selon la somme de leurs coûts.
Le solveur ajoute ensuite les objets un par un, tant que les budgets ne sont pas dépassés.

Objectif : obtenir une première solution simple et admissible, servant souvent de point de départ.

```
===== TEST 1 : Glouton Ajout (critère f_somme) =====

--- Glouton Ajout ? f_somme ---
  * Objet{ utilite=10, couts=[4, 3, 2] }
  * Objet{ utilite=8, couts=[3, 3, 3] }
Utilité totale = 18
```

Test 2 Glouton Ajout (critère : f_max)
Ce second test utilise un critère différent : f_max, qui prend en compte le coût maximal parmi les trois dimensions.
Objectif : montrer l'impact du comparateur sur la solution finale. Deux gloutons "ajout" peuvent produire des résultats différents.

```
===== TEST 2 : Glouton Ajout (critère f_max) =====

--- Glouton Ajout ? f_max ---
  * Objet{ utilite=8, couts=[3, 3, 3] }
  * Objet{ utilite=10, couts=[4, 3, 2] }
Utilité totale = 18
```

Test 3 Glouton Retrait
Dans ce test, le solveur commence avec tous les objets, puis supprime les moins intéressants selon le comparateur f_somme.
Une fois la solution admissible, il tente ensuite d'ajouter des objets supplémentaires selon f_max.

Objectif : tester une stratégie gloutonne plus globale, basée sur un retrait progressif plutôt qu'un ajout progressif.

```
===== TEST 3 : Glouton Retrait =====

--- Glouton Retrait ---
  * Objet{ utilite=8, couts=[3, 3, 3] }
  * Objet{ utilite=10, couts=[4, 3, 2] }
Utilité totale = 18
```

Test 4 Hill Climbing (t = 1)
Ce test applique la méthode Hill Climbing à partir de la solution issue du Test 1.
À chaque itération, on génère un voisin en retirant un objet et en essayant d'en ajouter un autre (t = 1).

Objectif : améliorer la solution gloutonne si un voisin possède une utilité totale plus élevée.

```
===== TEST 4 : Hill Climbing (t = 1, sans plateau) =====

--- Hill Climbing (t = 1) ---
  * Objet{ utilite=10, couts=[4, 3, 2] }
  * Objet{ utilite=8, couts=[3, 3, 3] }
Utilité totale = 18
```

Test 5 Hill Climbing avec déplacements sur plateau (plateau = 3)
Même configuration que le Test 4, mais le solveur est autorisé à faire jusqu'à 3 mouvements sans amélioration immédiate (plateau).
Cela permet parfois de franchir un palier et d'atteindre une meilleure solution locale.

Objectif : montrer l'effet des plateaux dans le processus d'optimisation.

```
===== TEST 5 : Hill Climbing (t = 1, plateau = 3) =====

--- Hill Climbing (t = 1, plateau=3) ---
  * Objet{ utilite=10, couts=[4, 3, 2] }
  * Objet{ utilite=8, couts=[3, 3, 3] }
Utilité totale = 18
```

========== PARTIE TESTS JUNIT ==========

Objectif des tests JUnit

En complément des tests exécutés via la classe Main et TestAll, le projet inclut également une série de tests automatisés réalisés avec JUnit 5. Ces tests permettent :

- de vérifier la validité des méthodes clés du projet
- de garantir que les solutions produites par les solveurs respectent bien les contraintes
- d'éviter les régressions lors des modifications du code.

L'utilisation de JUnit apporte une validation fiable et reproductible, indépendante de l'affichage console.

1 Test JUnit : Vérification de la méthode estAdmissible() (SacADos.java)

Le premier test JUnit porte sur la méthode estAdmissible() de la classe SacADos, qui permet de vérifier si une solution (une liste d'objets sélectionnés) respecte les contraintes budgétaires du sac à dos multidimensionnel.

Objectif du test

L'objectif est de s'assurer que :

- la méthode calcule correctement les coûts cumulés des objets

- les compare aux budgets disponibles
- et renvoie true si la solution est admissible

Ce test permet donc de valider la base du fonctionnement du modèle, avant l'application des algorithmes gloutons ou du Hill Climbing

2. Test JUnit 2 : Vérification de la méthode utiliteTotale() (SacADosUtiliteTest.java)

Ce test a pour objectif de vérifier que la méthode utiliteTotale() de la classe SacADos calcule correctement la somme des utilités des objets sélectionnés.

La méthode doit :

- parcourir la liste d'objets passée en paramètre,
- additionner leurs utilités,
- retourner la valeur totale.

Ce test est volontairement simple afin de valider le fonctionnement de base avant de tester des cas plus complexes.

3. Test JUnit 3 : Solveur Glouton Ajout (GloutonAjoutSolverTest.java)

Ce test vérifie le fonctionnement du solveur Glouton Ajout, dont le rôle est de sélectionner progressivement les objets les plus intéressants tant que les budgets ne sont pas dépassés. L'objectif est de s'assurer que l'algorithme respecte à la fois :

- l'ordre imposé par le comparateur (f_somme)
- les limites de budget
- et qu'il retourne bien une solution admissible

4. Test JUnit 4 : Solveur Glouton Retrait (GloutonRetraitSolverTest.java)

Ce test valide le fonctionnement du Glouton Retrait, la seconde variante gloutonne utilisée dans le projet. Contrairement au glouton par ajout, celui-ci :

1. commence par sélectionner tous les objets
2. retire progressivement les moins intéressants selon un premier comparateur
3. tente ensuite d'en réajouter selon un second comparateur.

L'objectif du test est de vérifier que cette logique est bien respectée et que le solveur retourne une solution **admissible**.

5. est JUnit 5 : Méthode Hill Climbing : amélioration d'une solution (HillClimbingSolverTest.java)

Ce test JUnit vérifie le comportement du solveur Hill Climbing, un algorithme d'optimisation locale destiné à améliorer une solution initiale en explorant son voisinage.

L'objectif principal de ce test est de vérifier que le Hill Climbing :

- n’aggrave jamais la solution initiale,
- et tente systématiquement de l’améliorer lorsque cela est possible.

-------==== CONCLUSION ====-------

Ce projet modélise de bout en bout un système complet de :

1. Simulation d’une équipe municipale création, évaluation et filtrage de projets.
2. Transformation en instance de sac à dos multidimensionnel contraintes budgétaires multi-coûts, utilité maximale recherchée.
3. Résolution heuristique glouton (ajout et retrait), optimisation locale avec Hill Climbing.

La structure modulaire (packages séparés : equipe, sacADos, solveur) rend le projet clair, extensible et facile à maintenir.
Les résultats montrent que les heuristiques permettent d’obtenir des solutions admissibles de bonne qualité tout en respectant les contraintes imposées.